

AD-A182 193

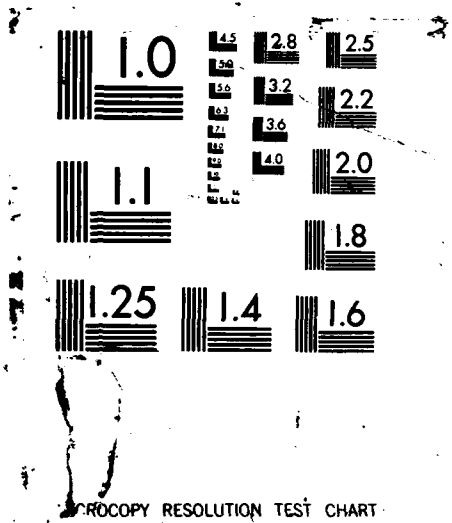
ON REAL-TIME OPERATING SYSTEMS(U) MARYLAND UNIV COLLEGE 171  
PARK DEPT OF COMPUTER SCIENCE S LEVI ET AL APR 87  
CS-TR-1838 N00014-87-K-0124

UNCLASSIFIED

F/G 12/5

NL

END  
8-87  
DTIC



## REPORT DOCUMENTATION PAGE

AD-A182 193

1b. RESTRICTIVE MARKINGS

N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT  
approved for public release;  
distribution unlimited.2b. DECLASSIFICATION/DOWNGRADING SCHEDULE  
N/A4. PERFORMING ORGANIZATION REPORT NUMBER(S)  
CS-TR-1838

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION  
University of Maryland6b. OFFICE SYMBOL  
(if applicable)  
N/A7a. NAME OF MONITORING ORGANIZATION  
Office of Naval Research6c. ADDRESS (City, State, and ZIP Code)  
Dept. of Computer Science  
University of Maryland  
College Park, MD 207427b. ADDRESS (City, State, and ZIP Code)  
800 North Quincy Street  
Arlington, VA 22217-50008a. NAME OF FUNDING/SPONSORING  
ORGANIZATION8b. OFFICE SYMBOL  
(if applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

N00014-87-K-0124

8c. ADDRESS (City, State, and ZIP Code)

10. SOURCE OF FUNDING NUMBERS

PROGRAM  
ELEMENT NO.PROJECT  
NO.TASK  
NO.WORK UNIT  
ACCESSION NO.

11. TITLE (Include Security Classification)

On Real-Time Operating Systems

12. PERSONAL AUTHOR(S)

Shem-Tov Levi and Ashok Agrawala

13a. TYPE OF REPORT  
Technical13b. TIME COVERED N/A  
FROM TO14. DATE OF REPORT (Year, Month, Day)  
April 198715. PAGE COUNT  
22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper reviews the requirements for a distributed fault tolerant real-time operating system. Various aspects, in addition to scheduling, are analysed to emphasize the importance of developing such operating systems separately, rather than modifying or building on top of regular ones.

DTIC  
ELECTE  
JUN 30 1987  
S E D

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT

☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS21. ABSTRACT SECURITY CLASSIFICATION  
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL



CS-TR-1838

April 1987

On Real-Time Operating Systems \*

Shem-Tov Levi and Ashok K. Agrawala

COMPUTER SCIENCE  
TECHNICAL REPORT SERIES



UNIVERSITY OF MARYLAND  
COLLEGE PARK, MARYLAND

20742

CS-TR-1838

April 1987

**On Real-Time Operating Systems \***

Shem-Tov Levi and Ashok K. Agrawala

Department of Computer Science  
University of Maryland  
College Park MD 20742



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

(\*) This research is supported in part by a contract from The Office of Naval Research to The Department of Computer Science, University of Maryland.

Contract No. N00014-87-K-0241

# On Real-Time Operating Systems

Shem-Tov Levi and Ashok K. Agrawala  
University of Maryland  
Dept. of Computer Science  
Computer Systems and Analysis Group  
College Park, MD 20742

14 November 1986  
Revised 22 April 1986

## Abstract

This paper reviews the requirements for a distributed fault tolerant real-time operating system. Various aspects, in addition to the scheduling, are analysed to emphasise the importance of developing such operating systems separately, rather than modifying or building on top of regular ones.

## 1 Introduction

### 1.1 What is an Operating System?

There are many definitions and approaches for stating what an operating system is. Peterson and Silberschats open their book [18] with the following statement:

*An operating system is a program which acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute programs. The primary goal of an operating system is thus to make the computer system convenient to use. A secondary goal is to use the computer hardware in an efficient way.*

Traditional operating systems have been built up with the above incentive. However, the development of microcomputers changed both the emphasis and the order of significance of the goals as listed above. Tanenbaum and Van Renesse [25] express it as follows:

*An operating system is a program that controls the resources of a computer and provides its users with an interface or virtual machine that is more convenient to use than the bare machine.*

It becomes even more complicated when we try to define what a distributed operating system is. Tanenbaum and Van Renesse ([25]) find *transparency* as the key concept:

A *distributed* operating system is one that looks to its users as an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).

Bayer et al. [3] describe the needs in an operating system as it has become more and more complex:

It is a principle of science that as complexity increases, the need for *abstractions* to deal with this complexity also increases. The evolution of operating systems is no exception. Early abstractions were files and processes. In each instance the abstraction takes the form of some non-physical resource and benefits both the system and the user.

...The abstraction of the system permits more efficient systems management of the central processors as well as indirectly contributing to the ease of management of all other resources.

Caspi and Halbwachs ([7]) try to focus on time dependent systems when presenting their model for temporal proof system:

Though time independent approaches present many advantages (portability, versatility, easy design and proof), they cannot apply to systems whose correctness is explicitly time dependent, that is, mainly, to two important classes of systems:

- So called "real-time" systems, which can be found in the field of industrial control: On one hand the specification of such systems contains real-time constraints (response times, sampling frequencies ...), and on the other hand, in order to meet these time constraints, the implementation of such systems often takes advantage of the knowledge of execution times, so as to save time consuming synchronisation mechanisms.
- Hardware systems, which implementation is also often optimised by taking into account the response times of the elementary devices, which can be quite precisely known.

Modern languages for real-time applications, as ESTREL [4], implement a large portion of classical operating system facilities, thereby achieving in some properties run time independence of any particular operating system. The motivation for having independence of operating systems, especially in the ESTREL case, is to achieve a verifiable system of timing properties. Using different methods and based on both static semantics and behavioral semantics, a transition system may be defined. For example, in [4] a set of structural conditional rewrite

rules ('a la Plotkin) is used. Such a transition system derives the proof system for the timing properties. The language is therefore totally synchronous, and the run time libraries are more of an integral part of the operating system. Moreover, side by side with achieving independence of operating systems, a strong dependency on the language is developed. The possibility of having a provable process written in a different language is lost. The dependency on the language becomes too strong. In [4], artificial statements should be added, since only explicit time-related instructions are allowed to consume time. Implicit time consuming process needs additional artificial explicit time consuming instructions to provide a basis for temporal reasoning. The result obtained is that run time libraries absorb a portion of the operating system, while reducing in some cases the generality of using more than one language, and the expectability of performance in cases of real systems in which synchronicity is not total.

More asynchronous languages, as ADA - US MIL-STD 1815A [1], implement task facilities and synchronisation between tasks, such that programmers don't have to deal with invoking system calls from lower level programs. In ADA only one type of synchronisation is supported, the rendezvous. If another type of synchronisation is required by the application program then services of the operating system should be invoked. In Unix ([6]) and many other operating systems, services of operating systems are given by a high level programs, and thereby the bridge between an operating system and language run time libraries is provided.

## 1.2 What is a distributed real-time system?

So far most of the real-time applications have used non-real-time operating systems, and have implemented their real-time properties within the application. Real-time operating systems, such as Data General's RTOS and Digital's RMX-22M, have been implemented.

However, these systems are real-time only in the sense that they support "hard" scheduling priorities; they do not support the notions of critical timing deadlines or fault tolerance. Furthermore, the above operating systems are not distributed operating systems, although some adaptations were made for network structure systems. The most common scheduling method, implemented in operating systems as the above, is through a user's "supervisor" program (or procedure). This supervisor, invoked by the timing hardware, maintains the priorities assigned to each task according to its own internal time counters. The operating system's dispatcher then sorts the queue accordingly. The supervisor itself is kept with the highest priority possible, such that other programs would not mask it easily.

An enormous effort was invested in developing "regular" distributed operating systems [25], whose main goal is maintaining a multiuser programming environment. The architectural differences between centralized and distributed systems emphasise issues, out of which some were of no concern when using

centralized operating systems.

The first issue is the communication, which plays an important role in a distributed architecture. Unlike centralized systems, and even unlike network oriented systems, the communication is needed for executing tasks that constitute the distributed environment. Examples for such tasks are remote procedural calls or migration of data and programs. The concurrent nature of a distributed system emphasises issues of sharing programs and data, such as mutual exclusion [21] deadlocks [8,14], termination [9,10,26,2] etc. Interprocess communication algorithms allow the composition of various degrees of atomicity [13] while maintaining concurrency and asynchronous behavior of the system. Many operating systems use the request/reply communication (Amoeba, Cambridge, V) with different sizes of packets, and some other systems are implementing communication which is closer to remote procedural call (e.g. Eden). However, most of the distributed operating systems known to us still fail to address problems which are of timing dependency nature. Two examples of such problems that are rarely solved with respect to real-time are the overhead requirements of preparing, sending and receiving, and the idempotency problem (guaranteeing that a remote call is executed only once).

The second issue which is important in a distributed operating system, even more than in a centralized one, is the naming mechanism and the way data is accessed. Various approaches for naming concepts and file services are found in today's operating systems. The differences are fundamental and govern the whole architectural concept of each of the systems. Protection mechanisms also differ from one system to another, in concept as well as in implementation architecture. Various aspects affect objects binding in a distributed environment. In particular, aspects concerning high level context of naming, or file systems [22], where directory management and context initialisation (binding) are much more complicated due to the difficulties in mapping objects that are distributed at the lower level.

The third issue that is significantly different in a distributed system is resource allocation. To begin with, processors in a distributed system are a resource one should allocate. One can find processor allocation policies that vary from fairly static ones (as in V or Cambridge processor bank) up to dynamic allocations (as in Amoeba's pool of processors). The optimal solution, as always, lies somewhere between maximising the parallelism, thereby achieving as much computation in parallel processors as possible, and minimising interprocessor communication, achieved by grouping as many related processes as possible at the same site.

Finally, fault tolerance issues are of major importance in distributed systems. A distributed operating system inherently has the potential to be more reliable than a centralized one, because the latter maintains only one instance of a critical component. Having distributed resource pools, one may maintain redundancy to a specific level in order to compensate for failure occurrences [12] due to hardware or software malfunctions. Moreover, in some cases one may

control "graceful degradation" of performance when failure is detected. Some existing distributed operating systems do not support fault tolerance issues (V). Most of them rely solely on reset and boot services (Cambridge, Amoeba), and only few support more sophisticated features, such as check-point update (Eden) etc. Fault tolerance features relate strongly to reconfiguration services that are supported by the operating system. The simplest service is the boot or login service, but further steps are needed to support reaction to failures, such as tracking the progress during execution etc.

Different approaches are found in various distributed operating system designs concerning the above issues. Most of the designs, even those where the system is built on top of a centralized one (as in Eden which is built on top of Unix), try to maintain the principle of transparency of resources used.

However, the need for emphasizing real-time issues in application support has spread. Examples of such applications are nuclear power plant control, industrial plant control, medical monitoring, digital fly-by-wire avionics and weapon delivery systems. All these systems are of real-time type, but as can be seen immediately, their nature requires high safety and reliability as well. Although real-time constraints and fault-tolerance requirements don't always coincide, these two disciplines should be both considered in many applications. The environment in which real-time system design takes place includes the processor, the operating system, the programming language with its compiler and run time libraries, the network structure in which a distributed computation takes place, and special aids used during the implementation phase. Each of the above components is of crucial importance to real-time properties, therefore designing real-time systems is thought to be one of the most complex programming activities [27]. In addition, real-time systems usually have strict reliability and safety requirements which severely increase the programming complexity. Furthermore, unlike non-real-time programs, real-time systems are implementation-dependent. Therefore, unlike traditional operating systems, a real-time operating system is extremely application-oriented. Its major task is to support real-time application programs, although programs with no timing constraints can be served as well. A real-time program that was executed successfully when implemented in one environment does not necessarily execute successfully in another environment. Changes in the computer hardware, communication network, operating system, or peripheral device response time may change the system's behavior to a point where it no longer satisfies a particular set of requirements. Usually it is not possible to modify a traditional operating system by adding a "timing - package" to obtain a real-time operating system, and the required properties such a system should have are listed later in this paper.

### 1.3 The Orthodox against the Liberal

Two key properties characterise the different nature of a real-time operating system with respect to a traditional one:

1. *Bounded time* - Each job should be finished within a bounded time. Furthermore, the relative progress of a set of tasks should be controlled and predictable.
2. *Design controllability* - The real-time operating system should allow (within limits) control to an application designer on the above.

Both properties are derived from two basic characteristics of the real-time environment: the need to execute programs within a *deadline* specified, and the high *dependency* on the application. The term "real-time" itself is application dependent. Dealing with a biological system may require a response time whose order of magnitude is seconds, whereas with electronic systems it might be micro-seconds or less. A liberal interpretation of the above two key properties finds them sufficient to categorise the class of real-time operating systems. The orthodox approach defines the required properties much further, and includes explicitly more features of the operating system.

Although run time libraries which implement "operating system" functions may serve equivalently in execution, this approach does not appear to show any performance advantages with respect to an operating system. A distributed system, where a total synchronicity is impossible and fault tolerance aspects are a major factor, emphasises the need for an operating system in the wide sense. In such a system, resource allocation should be supported in a way in which the above two key principles are guaranteed. However, one cannot ignore the needs for other services from the system. Moreover, unlike in a regular system, the performance with respect to real-time should be well defined for *any* service because of the nature of resource allocation. Even the case where only one of the jobs is a hard real-time job, and all the rest are not, is to be taken into account as a real-time problem, and proper allocation and load balance should be derived. As we show later in this paper, even naming and access control have a great influence on real-time performance of the system, and should be designed especially for real-time applications.

The orthodox approach might appear to be somewhat too traditional, yet it serves the basic incentive of defining the system in a way that prevents unexpected "surprises" during run time, as well as providing a verifiable tool with uniform behavioral properties.

## 2 Properties of a Fault Tolerant, Distributed, Real-Time Operating System

### 2.1 Resource Management / Allocation

In real-time operating systems the resource management and allocation mechanism should adhere to application requirements. In regular operating systems this mechanism is entirely application independent, and is maintained as an internal issue of the operating system.

#### 2.1.1 Scheduling

Servers that are shared between processes need queues that are manipulated by schedulers in order to set the order in which the service is given. In regular operating systems we find schedules for processing, for I/O services, for network services, etc. Many scheduling disciplines are possible, and each discipline may be suitable for scheduling a specific service. A particular scheduling policy for a particular service should adhere to the requirements of the jobs it serves. However, each scheduling discipline influences the implementation of an application very strongly. The strong interaction between a real-time operating system and the application programs implies that when a real-time application job is running, the system's global discipline state is relatively stiff. Yet, there are scheduling disciplines that a real-time system can never support. One example is a round robin discipline, in which the overhead of switching control between processes does not adhere to timing efficiency. The most general scheduling is divided into two parts:

- *Off-line scheduling*: a process in which the management of the policy of a discipline of a service scheduling is dynamically updated. For example, when a resource is added to the system, a new discipline may be adequate, as well as recognizing the availability of that resource.
- *On-line scheduling*: application of current discipline.

In order to describe how the scheduling should be based on real-time constraints, we have to define a model that describes both the scheduling process and the items we want to schedule – the computation processes. One should note that in a real-time operating system the order in which grants for service are given are such that timing requirements are met. The "hard" scheduling priority levels should be used as measures of emergency rather than as measures of urgency of execution due to real-time constraints.

Scheduling can be considered as maintaining ordered lists (queues), whose items are members of a given set, and the items are sorted according to a specified key. In our model, the given set is the set of active processes. From a real-time point of view, the set of active processes can be partitioned into two disjoint subsets:

1. Synchronous processes which are required to be activated regularly in a given particular frequency.
2. Asynchronous processes, each of which appears irregularly, but within a bounded frequency.

Each of the above processes, say  $P_i$ , is characterized with a *real-time constraint* expressed as a triple ([17])

$$\langle c_i, f_i, d_i \rangle$$

where  $c_i$  is the computation time of  $P_i$ ,  $d_i$  is its deadline and  $f_i$  is its frequency (for the asynchronous case it is the bound). Two important properties of the above model hold in case a working solution is feasible.

1.  $\forall i$  in the model:  $c_i < d_i < 1/f_i$ .
2.  $\sum_i c_i f_i \leq$  the number of available resources.

The first property must hold continuously, and is a consideration of the "on-line" scheduler, while the latter is a condition for the "off-line" scheduler. The latter requirement is stronger than what is really needed: it is based on the maximal demand rate the system can find due to asynchronous processes.

From the above model we can derive the property that distinguishes a real-time operating system from others:

- In a real-time operating system the scheduling decisions are based on the real-time constraints (the above triples) of the active processes in the system.

The following scheduling examples ([17]) demonstrate the meaning of the above statement.

**Examples:** Various algorithms may be applicable for an architecture that has a local queue for a specific resource. The most obvious scheduling algorithm is the "earliest deadline" algorithm. The scheduler chooses to execute the process whose deadline is the earliest to happen. A second approach is to use a scheduling policy which chooses the process whose maximal delaying possibility is the lowest. This approach is called the "least slack" algorithm, where the slack of a process at time  $t$  is defined as the maximum time which a run time scheduler can delay it, without disobeying the constraints.

$$slack(P_i, t) = \max(d(t) - t - c(t), 0)$$

A third example of real-time scheduling is the "latency" scheduling, in which one considers the whole global state of the system constraints as a graph, calculates the latency of the constraints, and allocates the "next-time-slice" to the system constraint with the appropriate latency.

The on-line scheduler is activated by process requests and by time servers. Its implementation should be very efficient, at "kernel" level, so that the overhead of management is minimized. The off-line scheduler can be implemented at a user process level and not necessarily in the kernel level. User processes are also allowed to be off-line (in other words their deadline can be  $\infty$ ), e.g. for log-in, compile, link and so on. As such, they can be served in any time independent discipline (LIFO, FIFO etc.), a policy which is used for ordering processes that have the same non-critical "latency".

### 2.1.2 Processor allocation

The allocation of a processor as a resource by the real-time scheduling algorithm might be governed by special processing constraints. Such an allocation might be in conflict with architecture-timing optimization. The next paragraph reviews one way of modelling and solving such situations.

Software tasks in time-critical real-time systems are usually divided into several threads, and each of the threads must satisfy an execution-time constraint, denoted as the port-to-port processing time. For example, in the Ballistic Missile Defense application, 23 tasks were divided into 7 threads. A model of the execution time of a thread can be constructed ([15]) of four components:

1. Execution time of the task on the processor ( $E_i$ ) which depends on the task size ( $T_i$ ) and the processor MIPs rate ( $\mu$ ):

$$E_i = T_i / \mu.$$

2. The communication network and operating system overhead ( $Ov$ ), which is used for concurrency control, integrity checking, recovery check-point update, etc.
3. Inter-processor communication ( $IPC$ ), whose cost is higher if communicants reside on different processors.
4. Waiting time ( $WT$ ), which is consumed when the task waits in the processor enablement queue. This figure depends highly on the sizes and number of tasks, the processor load, and the number of enablements. (Especially if large tasks are assigned to the same processor.)

Therefore,

$$E_T = \sum_i (E_i) + Ov + IPC + WT.$$

Enhancement of the performance of a system, as well as increasing the system's margin away from being unable to satisfy the demands, is done by minimizing  $E_T$ . For a given network,  $Ov$  and the number of enablements is relatively a constant. Hence, in order to reduce  $E_T$  the following steps should be adopted:

- Reduce *WT*: large tasks should be assigned to different processors.
- Reduce *IPC*: tasks with high IPC cost with each other should be assigned to the same processor.
- Reduce *E<sub>i</sub>*: large tasks should be assigned to processors with higher MIPS rate.

The above considerations yield the following sequence of activities in designing an allocation scheme:

- A set of constraints is determined, to reduce the waiting time and the task execution time. It may be performed in the following method:
  1. Information is entered to the model about the tasks (sizes, execution frequency of each task, number of data units exchanged between each pair of tasks) and the network (Inter-processor distance, constraints).
  2. Constraints are imposed on the model. In a matrix notation it is expressed as follows ([15]):
    - Task preference matrix: Certain tasks (out of  $m$ ) can be executed only on specific processors (out of  $n$ ). These restrictions are formulated as an  $m \times n$  matrix of 0's and 1's.  $\text{Element}(i, j) = 0$  means task  $i$  can't be assigned to processor  $j$ .  $\text{Element}(i, j) = 1$  means no restriction on task  $i$  with respect to processor  $j$ .
    - Task exclusive matrix: Defines mutually exclusive tasks, and expressed as an  $m \times m$  matrix.  $\text{Element}(i, j) = 0$  means no constraint between task  $i$  and task  $j$ .  $\text{Element}(i, j) = 1$  means task  $i$  and task  $j$  can not be assigned to the same processor.
- A cost function that measures the IPC cost is formulated.
- An algorithm that searches for the allocation with the minimum total cost is determined.

### 2.1.3 Architecture dependency

Architecture of resources should be a significant consideration when allocating them to processes. Distances between required resources may imply different execution time, and thereby may shrink the set of possible allocations within given timing constraints. Minimising the IPC to meet timing constraints can be considered using the above model. The cost function of the IPC depends on the following parameters:

- Task coupling factors  $c_{j,k}$ : number of data units transfered from task  $j$  to task  $k$ .

- inter-processor distance  $d_{q,p}$ : the cost of a transfer of one data unit from  $q$  to  $p$ .

The allocation in matrix notation is expressed as  $\text{element}(j, p) = 1$  when task  $j$  assigned to processor  $p$ , and zero otherwise.

The algorithm to derive a proper resource allocation might use the task preference and the task exclusive matrices, and the minimization of the IPC cost uses both the task coupling factors and the inter-processor distances. In [15] a branch and bound technique is considered, using a two phase algorithm: setting and back-tracking.

## 2.2 Time Services

In addition to simple time services, which are found also in "regular" operating systems (e.g. Get time, Get date, Time stamp), some very complex services are needed in a real-time operating system. One example is the clock-reset service, which is important as an inter-process service (for physical synchronisation) as well as an internal service for a process. A time service in a distributed system uses an algorithm that keeps a collection of clocks locally monotonic (between updates), synchronised and adequately accurate with respect to some time standard. Keeping the clocks locally monotonic is simple, and can be achieved with logical clocks that obey some rules ([11]), but this solution arises both anomalous behavior and a large drift between different clocks. In order to examine the adequacy of a service, we have to define a model that describes it.

- Let  $C_i(t)$  denote a function that maps real time to clock "i" time.
- A *standard* clock "i" is one with  $\forall t : C_i(t) = t$ .
- A clock "i" is *correct* at time  $t_0$  if  $C_i(t_0) = t_0$ .
- A clock "i" is *accurate* at time  $t_0$  if the first derivative of  $C_i(t)$  is 1 sec/sec at  $t_0$ .

The use of a set of physical clocks  $\{C_i(t)\}$  improves the system's performance, with respect to drift, up to a limit. However, applying some synchronisation procedure is required. This procedure can be regarded as

$$C_i(t) \leftarrow F(C_{i1}(t), C_{i2}(t), \dots, C_{ik}(t))$$

where the function calculates its result from a distributed data. L. Lamport shows in [11] that with physical clocks, and with a set of processes that form a strongly connected communication graph whose diameter is  $d$ , and with a network in which the message delays are bounded, bounds for clock accuracies and for clock correctnesses may be derived. In particular, each clock accuracy is bounded by

$$\forall t : \left| \frac{dC_i(t)}{dt} - 1 \right| < \delta_i \ll 1.$$

The correctness of each clock is obtained by sending messages from one clock to another every  $\tau$  seconds, with an unpredicted message delay that is smaller than  $\eta$ , and the bound is

$$\forall i : \forall j : |C_i(t) - C_j(t)| < d(2\delta\tau + \eta)$$

for all  $t$ .

Marsullo and Owicki, in [16], consider two additional synchronisation functions. Their work makes a use of the local knowledge of the errors of the local clock. Suppose every clock "knows" it is correct within the interval  $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$ . Hence,  $E_i(t)$  is a bound on the "i" clock maximal error, a quantity it is able to calculate by combining the error sources that can be expressed by the following model ([16]):

- The error that comes into effect right on the clock *reset*, such as discretisation and other constant errors.
- The *delay* between the time this clock is read until another clock uses this readout for its reset.
- The *degradation* of time counting that develops between consecutive resets.

According to this model, we can define a property that is weaker than correctness. A time service is *consistent* if the intersection of the intervals, defined by the set of its clocks  $\{C_i(t)\}$ , is nonempty. For example, in case there are two clocks in the system time service, then consistency means

$$|C_i(t_0) - C_j(t_0)| \leq E_i(t_0) + E_j(t_0).$$

Two synchronisation algorithms are presented in [16], using the above error source model. The incentive is to obtain a time service that is at least as accurate as the best clock in the system.

1. *Minimize maximum error* is an algorithm that obeys two rules:

- Upon receiving a time request it answers with its current  $C_i(t)$  and its current  $E_i(t)$ .
- At least once every  $\tau$  time-units each clock requests the time from its neighbor clocks. Any response which is inconsistent with its own readout is ignored. In case the response is consistent, and the error of the just-received response (combining both the error upon its being sent and a bound on possible additional drift) is smaller than the local one, then the response overwrites the local readout.

The update results in a smaller error, since it is constructed from the smallest clock error in the system and the error generated since the last reset. Examining the error obtained, one can see that the "long term"

error growth is due to the drift of the most accurate clock in the system. In some ways, the goal of having a service as accurate as the best clock in the system is achieved, but one should recall two flaws in the above algorithm.

First, the algorithm is based on having valid upper bounds on the clock's drift rates. (Without these bounds, or in other words using infinite intervals, the algorithm is of the same order as [11]). Secondly, the case in which a clock finds itself inconsistent with all the rest is not regarded.

2. *Intersection of time intervals* is also an algorithm that consists of two rules:

- Upon receiving a time request it answers with its current  $C_i(t)$  and its current  $E_i(t)$  (exactly as in the previous algorithm).
- At least once every  $\tau$  time units each clock requests the time from its neighbor clocks. Each clock collects all the answers and calculates an interval from the highest lower edge and the lowest higher edge. If it is consistent, then the clock is reset to the resulting interval. The error of the resulting interval is smaller or equal to the smallest in the system.

This algorithm originates in the logical assertion that if a formula  $p$  is true over the period  $T_1$ , and a formula  $q$  is true over the period  $T_2$ , then the formula  $p \wedge q$  is true over the intersection of  $T_1$  and  $T_2$ , if they do overlap. Although the above assertion is too simplified for a general deductive system, since there are causality relations in which the consequence appears only after the cause is finished, it is accurate for clock synchronisation. This algorithm achieves better (small) errors than the previous one, based on the fact that the overlap of two intervals must be smaller than or equal to the smallest between the two. Yet, it still comes short due to the requirement of having a correct upper bound for every clock's drift rate. Another weakness is that this algorithm is also less fault tolerant than the first one, since there are consistent states recovered by the first that the latter ignores. Such a trivial example is the case where only one readout is bad enough to cause inconsistency to be obtained.

In [16] an interesting approach of applying the same idea to rates as to intervals (a "velocity" feedback) is mentioned. We find statistical treatment (as in confidence intervals) even more interesting. To conclude this section, recall that the complexity of a time service in a distributed fault tolerant real-time operating system, and the accuracy requirements emphasize the need of having an operating system solution to these requirements.

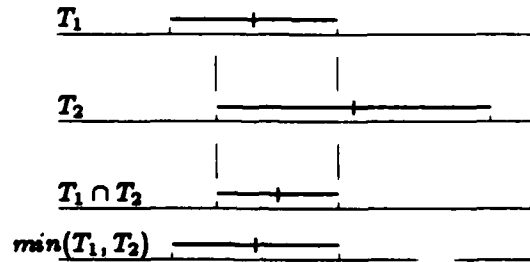


Figure 1: The two clock synchronisation algorithms.

## 2.3 Communication

In addition to functional requirements of communication, as supported by any operating system, the two key properties mentioned above (bounded time for jobs and design controllability to application designers) should be maintained by the communication support of a real-time operating system.

### 2.3.1 Message Passing

The *type of model* used (e.g. Master/slave, balanced, centrally controlled, Post-Office like, etc.) influences strongly both fault tolerance issues and real-time constraint issues. In cases where communication is more synchronous, processes have to poll and wait for their turn to send messages. For example, consider a centrally controlled communication network, in which one process dictates to all others when each is allowed to transmit to a network. Such a system is often found in a unibus network (e.g. the Mux-Bus), and it reduces significantly real-time performances, unless a special hardware (buffers and communication control) is added to each processor.

*Parameter passing* in remote procedure calls emphasises the interaction between the real-time properties of the communication network and the distributed solution implemented. Consider the reference versus value parameter passing. Using reference parameters assures a better concurrency control in cases of data sharing, yet it complicates the execution, thereby imposing longer access times. Using value parameters requires additional control to allow sharing, treating the block of data to be shared as a resource, but execution time is well defined for a specific allocation. Furthermore, in case the shared object is a program, "value" transfer allows concurrent execution of the replica. The above considerations suggest that in a real-time system, parameter passing for remote procedural calls should use value parameter-transfer in addition to a mutual exclusion control. In addition, different formats of information should be excluded, such that no irregular addition of "translation time" would be added to the execution time of a job. The above considerations influence the storage and access management, and even the issue of file structures.

*Fault tolerance of communication*, has a tremendous effect on timing bounds for messages. However, the communication network must adhere to the *correctness* of timing requirements of the application ([23,24]), as well as supporting failure modes. For example, consider the error model we introduced for a synchronisation of a distributed physical clock system. In order to have a reasonable estimate for  $E_i(t)$ , one has to have a reasonable bound for the communication duration, otherwise the time from the moment a readout has been sampled, to the moment it is used to reset a clock, is not well defined. These aspects are expanded in the next section.

### 2.3.2 Error Handling

The ability of the communication subsystem to handle errors and failures is an important measure of a system's robustness. Two possible error types should be handled: errors that originate in hardware failure (either a site failure or a communication network failure), and errors due to the design. While the first can be treated via redundancy and checks, the latter has to be treated with special techniques to allow detection and diagnostics of the problem. Some of the potential error detection and recovery procedures in a real-time systems has a lot to do with the communication architecture and activities, emphasising the special considerations due to real-time constraints.

When a remote site *crashes*, the use of a timeout mechanism in order to prevent endless waiting is found in some communication sub-systems of regular operating systems. In a real-time environment it should be noted that late delivery of a message may violate timing constraints of execution, and thereby may lead to a system crash. Therefore, possible timeout retries should be taken into account prior to scheduling in a real-time operating system, when the computation time is considered.

*Acknowledgements* are commonly used to obtain a safer communication protocol. Yet, communication duration thereby grows along with the network load. The trade-off between high fault tolerance and execution times should be one of the controllability features given to the application designer. Choosing between real-time performance and fault tolerance performance might be strongly influenced by the possibilities currently available.

*Idempotency* is a problem in any distributed construction. The requirement to allow transmission repetitions due to communication recovery procedures affects other desired system properties. For example, consider the use of a shared counter. Suppose a process has sent an increment command to this counter, and failed to receive an acknowledgement from the counter. If the recovery procedure sends another increment command, the result may be an increment by two. However, using some kind of "commit" protocol might overload the network, and naturally increase execution time of jobs.

*Channel properties* affect many other system properties and architecture issues. Some examples are: reordering of messages, message loss probability,

bounded delivery time, etc.

### 2.3.3 Issues of efficiency in implementation

Some implementation issues might affect the real-time and fault tolerance efficiency parameters of the system.

- The effects of reducing the *overhead* of the communication system to a minimum might enhance real-time parameters, whereas fault tolerance might be degraded.
- The usage of *special purpose* short messages (e.g. for remote synchronisation) may complicate the communication software and hardware, while time consumption can be reduced, compared to sending empty packets with such message types.
- The question whether to have *end-to-end acknowledgement* or not, can be considered as a trade-off between robustness of intermediate nodes versus end-to-end acknowledgement of delivery.

## 2.4 Name servers

Name service usually provides the mapping for three name spaces, keeping it unique for all three ([19]):

1. Character *string* names: used in the file system and in user programs.
2. Segment *numbers*: used by a running process to refer to an active segment.
3. A known segment table (of a user) provides physical *addresses* for the page table of a corresponding segment.

Each name is always interpreted with respect to a particular *context* ([22]). The context is a set of bindings of names to objects. Context initialising (also called binding, or linking, or loading) is the bridge between the high level human oriented names, to the low level machine oriented addressing world. A context initialising procedure usually consists of resolving and installing actions. The resolution of a name involves searching an existing object that is identified by a given name in a given context. The unknowns in such a search disqualifies it as a possible run time procedure in a real-time system. In a real-time system the context initialiser must be allowed only as an off-line task, probably executed before running a real-time job.

A file system (a high level context) in a distributed real-time operating system should support:

- human oriented names,
- multiple users,

- selectively shared contexts,
- the ability to distinguish user's intent from the programmer's,
- and the ability to adhere to a given timing constraint.

The result of this requirements ([22]) is a naming service that includes the following.

- A multiple directory system.
- A naming network, in which directories appear as named objects in other directories.
- Usage of a mechanism (closures) that connects an object that refers to other objects by name, with the context in which those names are bound. The simplest implementation of such a mechanism is through the usage of a directory as such.

In addition, in order to be able to adhere to timing constraints, the response time of a name service should always be reasonably bounded.

Name servers might be limited to small-size unique names, in order to reduce search time. Such a restriction arises issues of reusability of names, recalling the need to maintain the uniqueness of names in a given context, especially regarding the selectively shared contexts objective. However, implementing such a service correctly, allows any user to disconnect itself from external objects, but requires some additional tools for deletion of objects because other users may be linked to them.

The search rules in a name service in a real-time operating system should be efficient from the real-time point of view. As such, a direct entry in a directory should point to an object, and an indirect entry should specify the whole access path of the shared context to the closure in concern. Applying such an approach allows limiting the number of directories that should be searched for an access; for example, the user's working directory, a language library and a system library. The language and system libraries can be kept ordered and balanced, and thereby reduce search time to a minimal bounded time.

## 2.5 Data Access Strategy

### 2.5.1 Protection

Real-time embedded systems usually neglect protection mechanisms. However, a real-time operating system cannot. The real-time requirements, being the dominant ones, necessitate keeping protection timing cost as low as possible. For example, consider the issue of users being given direct access to required resources on remote sites. The access time is reduced, hand in hand with increasing the availability of a potentially restricted information. Protection systems

are divided into two categories: list oriented systems and ticket oriented systems (as capabilities). Access control lists imply a search procedure that is not adequate for real-time systems. Capability system, in which authorisation is a prior phase to run time, provides a better real-time environment.

### 2.5.2 Remote storage and directory services

In a distributed system an application program does not have an explicit knowledge about storage locations it uses. Therefore, a service which maintains a link to remote sites must be given. Furthermore, timing properties of different allocation instances may differ significantly. Therefore, the efficiency of such a service is very significant.

## 2.6 Fault Tolerance

It is not just desirable but may be necessary to support both safety (guarantee of not happening) properties and reliability (guarantee of happening) properties.

- Different modes of operation may be applicable in case of a node *failure*. During a recovery procedure, one may maintain a backup mode, as long as the recovery is not complete. The whole system might be reconfigured for such a procedure.
- *Communication failure* may require reconfiguration of the system to increase link robustness.
- *Redundancy*: Data and process replication may be needed to support a resilient computation. Reconfiguration schemes have to be devised when switching to any backup mode. A redundant execution of a process consumes a large amount of resources, and thereby slows down the system's real-time performance. A solution which is less time consuming, is to maintain the backup program in a standby mode, and activate it only upon failure. This solution requires that a failure will be detected early enough, to provide recovery procedure time, even in degraded performance mode. It also requires a continuous check-points update, to provide a starting point to the backup process.
- *Graceful degradation*: As parts of the system fail, it may no longer be possible to satisfy all the requirements of the application. Techniques for graceful degradation may be very useful to ensure that critical activities do not fail.

## 2.7 Other Services

Other services which might be offered to application programs are important as well.

### **2.7.1 Service architecture**

Determining the way in which application programs use the system calls / services has an enormous effect on its real-time performances. There are many possible architectures to implement services. For example, a service may poll continuously, waiting for a request to appear, or it may be awakened as a classic procedure. Another example is to have a service treated as a resource, and in case more than one user requests it, to maintain a queue for it. The latter approach is not adequate for real-time applications, because it forces an uncertain execution time into a job.

### **2.7.2 Reconfiguration services**

This service is very important in a dynamic system. Sites which are turned on should boot themselves, sites that fail must reboot themselves, and sites that are turned off should be detected by the resource management and allocation task. Moreover, in case of fault detection and recovery procedure, the reconfiguration is of extreme importance.

### **2.7.3 I/O device services**

I/O services are used in real-time as well as in regular operating systems. However, device speeds may significantly influence application performance, especially in jobs in which such services are in series with other timing constraints.

## **3 conclusion**

Summarising the properties required from a real-time fault tolerant distributed operating system, the following list is obtained.

1. Allocation of resources and scheduling of jobs should be in accordance with the bounded time given to execute the job. Furthermore, the relative progress of a set of tasks should be controlled and predictable.
2. Control (within limits) should be allowed to an application designer on the above. For example, in cases where timing efficiency and protection might reach a conflict, the application designer should be able to decide what a trade-off to make.
3. Time services should be given within specified accuracy and correctness.
4. Communication should support a reliable data transfer, while satisfying real-time constraints imposed.
5. Remote procedural calls should avoid migrations as possible, and pass value parameters side by side with maintaining a mutual exclusion support for shared objects.

6. Communication acknowledgement should rely on an intermediate robustness, instead of waiting for an end-to-end acknowledgement.
7. Naming network should be used, based on a multiple directory file system, using closures for shared objects.
8. Context initialisation (binding) should be allowed only in off-line jobs.
9. Protection mechanism should be "ticket" oriented.
10. Redundant backup jobs should use check points and be activated only when a failure occurs. Data replica should be updated via check points, and migrate upon failure, as an off-line recovery procedure.
11. A reconfiguration service should be activated both upon request (failure, login, boot, etc.) and periodically (turn off, etc.).

In general, all "unknown" parameters in the execution time of a job should be avoided, allowing them to occur only under an off-line discipline.

## References

- [1] *Reference Manual for The Ada Programming Language*, U.S. DOD (ANSI) MIL-STD 1815a-1983, Feb 1983.
- [2] Arora R., Rana S. and Gupta M., *Distributed Termination Detection Algorithm for Distributed Computations*, Inf. Proc. Letters, Vol 22 No 6 pp 311-314, May 1986.
- [3] Bayer R., Graham R. and Seegmuller G. (editors), Flynn M., Gray J., Jones A., Lagally K., Opderbeck H., Popek G., Randell B., Saltzer J. and Wiehle H., *Operating Systems: An Advanced Course*, Springer - Verlag, Berlin, Germany, 1979.
- [4] Berry G., Cosserat L., *The ESTREL Synchronous Programming Language*, Ecole Nationale Supérieure des Mines de Paris, France, March 1986.
- [5] Bochmann G., *Distributed Systems Design*, Springer-Verlag, Berlin Germany, 1983.
- [6] Bourne S., *The UNIX System*, Addison - Wesley Publishing Co., London England, 1983.
- [7] Caspi P., Halbwachs N., *A Functional Model for Describing and Reasoning Time Behavior of Computer Systems*, Acta Informatica, Vol 22 No 6 pp 595-628, March 1986.

- [8] Chandy K., Misra J. and Haas L., *Distributed Deadlock Detection*, ACM Transaction on Computer Systems, Vol 1 No 2 pp 144-156, May 1983.
- [9] Dijkstra E., Scholten C., *Termination Detection for Diffusing Computation*, Inf. Proc. Letters, Vol 11 No 1 pp 1-4, Aug 1980.
- [10] Dijkstra E., Feijen W., Van Gasteren A., *Derivation of a Termination Detection Algorithm for Distributed Computation*, Inf. Proc. Letters, Vol 16 pp 217-219, June 1983.
- [11] Lamport L., *Time, Clocks and Ordering of Events in a Distributed System*, Communications of the ACM, Vol 21 No 7 pp 558-565, July 1978.
- [12] Lamport, L., R. Shostak, and M. Pease, *The Byzantine Generals Problem*, ACM Trans. on Prog. Lang. and Systems, Vol. 4, no. 3, July 1982, pp. 382-401.
- [13] Lamport L., *On Interprocess Communication* parts I and II, Distributed Computing, Vol 1 No 2 pp 77-101, Springer-Verlag 1986.
- [14] Levi S., Plateau B., *A Distributed Algorithm for Deadlock and Termination Detection of Distributed Computations*, University of Maryland technical report CS-TR-1750, University of Maryland, Department of Computer Science, December 1986.
- [15] Ma P., Lee E., Tsuchiya M., *Design of Task Allocation Scheme for Time Critical Applications*, IEEE Proceedings - Real Time Systems Symposium, Miami Beach FA, Dec 1981.
- [16] Marsullo K., Owicki S., *Maintaining the Time in a Distributed System*, ACM Operating Systems Review, Vol 19 No 3 pp 44-54, July 1985.
- [17] Mok A., *Fundamental Design Problems for the Hard Real Time Environment*, MIT Ph.D. Dissertation, Cambridge MA, May 1983.
- [18] Peterson J., Silberschatz A., *Operating System Concepts*, Addison - Wesley Publishing Co., Reading Ma, July 1984.
- [19] Popek G., *Issues in Kernel Design*, (in *Operating Systems: An Advanced Course*, Bayer R., Graham R. and Seegmuller G. - editors), Springer - Verlag, Berlin, Germany, 1979.
- [20] Quirk W. (editor), *Verification and Validation of Real Time Software*, Springer-Verlag, Berlin Germany, 1985.
- [21] Ricart G., Agrawala A., *An Optimal Algorithm for Mutual Exclusion in Computer Network*, Communication of the ACM, Vol 23 No 1 pp 9-17, Jan 1981.

- [22] Saltzer J., *Naming and Binding of Objects*, (in *Operating Systems: An Advanced Course*, Bayer R., Graham R. and Seegmuller G. - editors), Springer - Verlag, Berlin, Germany, 1979.
- [23] Shankar A. U., Lam S.S., *Time Dependent Communication Protocols, Tutorial: Principles of Communication and Networking Protocols*, S. S. Lam (ed.), IEEE Computer Society, 1984.
- [24] Shankar A. U., Lam S.S., *Construction of Sliding Window Protocols*, CS-TR-1647 Computer Science Department, University of Maryland, Feb 1986.
- [25] Tanenbaum A., Von Renesse R., *Distributed Operating Systems*, ACM Computing Surveys, Vol 17 No 4 pp 419-470, Dec 1985.
- [26] Topor R., *Termination Detection for Distributed Computation*, Inf. Proc. Letters, Vol 18 pp 33-36, Jan 1984.
- [27] Wirth N., *Toward a Discipline of Real Time Programming*, Communications of the ACM, Vol 20 No 8 pp 577-583, Aug 1977.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is an Operating System? . . . . .	1
1.2	What is a distributed real-time system? . . . . .	3
1.3	The Orthodox against the Liberal . . . . .	6
<b>2</b>	<b>Properties of a Fault Tolerant, Distributed, Real-Time Operating System</b>	<b>7</b>
2.1	Resource Management / Allocation . . . . .	7
2.1.1	Scheduling . . . . .	7
2.1.2	Processor allocation . . . . .	9
2.1.3	Architecture dependency . . . . .	10
2.2	Time Services . . . . .	11
2.3	Communication . . . . .	14
2.3.1	Message Passing . . . . .	14
2.3.2	Error Handling . . . . .	15
2.3.3	Issues of efficiency in implementation . . . . .	16
2.4	Name servers . . . . .	16
2.5	Data Access Strategy . . . . .	17
2.5.1	Protection . . . . .	17
2.5.2	Remote storage and directory services . . . . .	18
2.6	Fault Tolerance . . . . .	18
2.7	Other Services . . . . .	18
2.7.1	Service architecture . . . . .	19

END

8-87

DTIC